# Automated Verification of Higher-Order Masking in Presence of Physical Defaults

Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert

# Automated Verification of Higher-Order Masking in Presence of Physical Defaults

Gilles Barthe[1], Sonia Belaïd[2], Gaëtan Cassiers[3], Pierre-Alain Fouque[4], Benjamin Grégoire[5], and Francois-Xavier Standaert[3]

[1] MPI for Security and Privacy and IMDEA Software Institute
[2] CryptoExperts
[3] Université Catholique de Louvain
[4] Université de Rennes
[5] Inria Sophia-Antipolis Méditerranée

**Abstract.** Power and electromagnetic based side-channel attacks are serious threats against the security of cryptographic embedded devices. In order to mitigate these attacks, implementations use countermeasures, among which masking is currently the most investigated and deployed choice. Unfortunately, commonly studied forms of masking rely on underlying assumptions that are difficult to satisfy in practice. This is due to physical defaults, such as glitches or transitions, which can recombine the masked data in a way that concretely reduces an implementation's security.
We develop and implement an automated approach for verifying security of masked implementations in presence of physical defaults (glitches or transitions). Our approach helps to recover the main strengths of masking: rigorous foundations, composability guarantees, automated verification under more realistic assumptions. Our work follows the approach of (Barthe et al, EUROCRYPT 2015) and thus contributes to demonstrate the benefits of language-based approaches (specifically probabilistic information flow) for masking.

**Keywords:** Side-Channel Attacks, Masking Countermeasure, Physical Defaults, Glitches, Automated verification, Composability, maskverif.

## 1 Introduction

While the design of cryptographic algorithms such as block ciphers is a relatively well-understood problem [26], the secure implementation of such algorithms is still a quite open topic. For example, the last two decades have shown that a wide range of side-channel attacks can be performed against cryptographic implementations, exploiting physical sources of leakage such as timing [27], power consumption [28] or electromagnetic radiation [20]. If no attention is paid, measuring such physical information enables retrieving cryptographic keys extremely efficiently. As a result, various types of countermeasures have been introduced to mitigate side-channel leakages, ranging from heuristic to more formal.

In general, checking that an implementation is protected against side-channel attacks is a complex and error-prone process (see [29] for an overview). As a result, countermeasures relying on a more established theory have gained in relevance over the last years, in order to simplify both the design and the evaluation of protected implementations. The masking countermeasure (which consists in performing the sensitive computations on secret-shared data) has been shown to be a particularly interesting candidate in this landscape [11]. The main reason is that practical security against physical leakages via masking can be reduced (under some noise and independence assumptions) to a much simpler (and abstract) security model, where the adversary just observes intermediate values during execution of the implementation [14]. We will next refer to this simpler abstract model as *ISW model*, after its inventors [25].

One advantage of the ISW model is that its conceptual simplicity makes it amenable to formal verification. This has been demonstrated in a series of works, including [31, 6, 17, 2, 3, 13, 34, 9, 10]. The most immediate benefit of formal verification is its automation, allowing to deal with the combinatorial complexity of proving masked implementations secure. This complexity is specially significant for implementations where secrets are split into a large number of shares; we call such implementations higher-order. Perhaps more importantly, formal verification has also been instrumental for advancing the state-of-the-art in masking. First, formal verification tools have been used to reduce the randomness cost of existing schemes. Second, strong non-interference, which solves a long-standing problem of compositional reasoning for masking, has first emerged in the context of formal verification, before being adopted in the literature on masking.

However, and to the exception of [9, 10], the abstractions in these tools still do not prevent the risks due to specific physical defaults that may happen when trying to implement masking in hardware or software devices. In fact, many masking schemes that are secure in the abstract ISW model become insecure (or at least less secure) when concretely implemented.

This is usually due to physical defaults which contradict the independence assumption required for secure masking. For instance, *glitches* are a form of physical default occurring when the information does not propagate simultaneously throughout execution. They introduce dependencies between the leakage of an instruction and of its predecessors (in the sense of dataflow analysis). These dependencies may cause hardware implementations proved secure in the ISW model to be practically vulnerable against side-channel attacks [30]. *Transitions* are another example of physical default which more typically happen in software implementations, where the value in a register is overwritten by another value and leads the leakages to depend on both [12, 1].

As a consequence, it is necessary to develop models and verification methods for proving security in presence of physical defaults. Bloem et al. [9] and Faust et al. [19] independently extend the ISW model in order to capture physical defaults such as glitches — we will next denote this model as the *ISW model with glitches*. In addition, Bloem et al. propose an automated method based on an estimation of Fourier coefficients for proving that an implementation is secure

in their model. They also use their verification method on a set of examples, including the S-Boxes of the AES and Keccak. Due to the computational cost of their approach, the tool primarily applies to the first-order setting, where secrets are split into two shares. Moreover, their method does not consider strong non-interference, which is key to verify complete implementations. By contrast, Faust et al. provide a hand-made analysis of some masking gadgets and prove their strong non-interference with glitches for arbitrary number of shares (at the cost of higher randomness requirements), and discuss simple conditions under which the composabiliy rules from [3] apply to implementations with glitches. As for [10], it is built on top of this submission (from on an earlier version of the current paper) and is still restricted to the verification of probing security only.

**Contributions.** We implement an efficient method for reasoning about security of higher-order masked implementations in presence of physical defaults. Our method follows a language-based approach based on probabilistic information flow for proving security of masked implementations [2], and so provides further evidence of the benefits of language-based approaches.

As in [2], our method follows a divide-and-conquer approach, embodied in two algorithms. Our first algorithm checks if leakages are independent of secrets for a fixed admissible set of observations. The algorithm repeatedly applies semantic-preserving simplifications to the symbolic representation of the leakages, until it does not depend on secrets or it fails. One significant improvement over [2] is that our algorithm (i) is sound and complete (no attack missed and no false negative) for linear systems of equations; (ii) it minimizes false negatives for non-linear cases. Our second algorithm explores all admissible observation sets, calling the first algorithm on each of them. This algorithm is carefully designed to minimize the number of sets to explore, using the idea of large sets from [2]. One significant improvement over [2] is that our algorithm (i) minimizes the number of large sets (ii) uses more sophisticated data structures that improve overall efficiency.

In addition, both algorithms are specifically tailored to a rich programming model, which we introduce to maximize applicability. The critical feature of our new programming model is that all instructions are annotated with a symbolic representation of leakage. Our programming model neatly subsumes several models from the literature, including the ISW model, the recently proposed ISW model with glitches, and a version of the ISW model with transitions. Moreover, our tool applies to three main security notions: probing security, threshold non-interference, and strong non-interference (which is essential for compositional reasoning). Our coverage of models and properties is displayed in Table 1.

We implement our method on top of `maskVerif` and evaluate our tool on existing benchmarks. Our tool is able to verify programs efficiently for security notions that bring stronger compositional guarantees than [9] and faster than state-of-the-art tools. For instance, checking probing security for the ISW multiplication at order 4 (resp. order 5) takes 1 (resp. 45) second using [2], while it takes only 0.1 (resp. 2.6) second with our tool. And checking probing security with glitches for DOM Keccak S-box at order 3 takes only 0.49s when it takes

**Table 1.** Verification of higher-order masked implementations in the ISW model (1), the ISW model with glitches (2), and a version of the ISW model with transitions (3)

| Tools | probing security | | | threshold non-interference | | | threshold strong non-interference | | |
|---|---|---|---|---|---|---|---|---|---|
| | (1) | (2) | (3) | (1) | (2) | (3) | (1) | (2) | (3) |
| `maskVerif` [2, 3] | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |
| Bloem *et al.* [9] | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| this work | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

more than 25min in [9] Note that we do not compare our results with [10] as the authors based their method on an earlier version of this submission.

## 2 Motivating Examples

Consider the logical `and`, which takes as input bits $a$ and $b$ and produces as output a bit $c$ such that $c = a \times b$ (we use arithmetic notation). A (first-order) masked implementation of this algorithm takes as input bits $a[0]$, $a[1]$, $b[0]$ and $b[1]$, called input shares, and outputs bits $c[0]$ and $c[1]$, called output shares. We consider two families of masked implementations and outline their verification. The first family is intended to provide protection against glitches. The second family is intended to provide protection against transitions.

### 2.1 Glitches

Figure 1 introduces a first-order masked implementation of logical `and` from [22] in an idealized hardware language. The program is given as a sequence of assignments. The instruction $r \leftarrow_{\$} \{0, 1\}$ is a random assignment, i.e. $r$ is sampled uniformly from $\{0, 1\}$. The assignments $t_2 \leftarrow_{\mathsf{ff}} t_1$ and $t_6 \leftarrow_{\mathsf{ff}} t_5$ are flip-flop assignments; they are used to store stable computations (so have no computational content), and stop the propagation of leakage. The remaining instructions are standard arithmetic assignments. The masked implementation must satisfy:

**correctness:** the masked implementation coincides with the original algorithm, i.e. $c = a \times b$, with $a = a[0] + a[1]$, $b = b[0] + b[1]$, $c = c[0] + c[1]$;

**security:** leakage does not reveal information about secrets. We make the definition of leakage precise below and sketch a proof of security.

We first consider correctness. We symbolically execute the program to compute for each program point an expression over input shares $a[0], a[1], b[0], b[1]$ and probabilistic variable $r$, see the middle column of Figure 1. We use $b[i] \times a$ as shorthand for $b[i] \times a[0] + b[i] \times a[1]$.

We now turn to security. We first define a symbolic representation of leakage, shown in the third column of Figure 1. The representation assigns to each program point a tuple of expressions over input shares $a[0], a[1], b[0], b[1]$ and probabilistic variable $r$. Random assignment $r \leftarrow_{\$} \{0, 1\}$ leaks singleton $\{r\}$.

Flip-flop assignments $t_2 \leftarrow_{\mathtt{ff}} t_1$ and $t_6 \leftarrow_{\mathtt{ff}} t_5$ leak singletons $\{b[1] \times a[0] + r\}$ and $\{b[0] \times a[1] + r\}$, i.e. the expressions they compute. Arithmetic assignments propagate transient leakages (due to glitches). For instance, assignment $t_1 \leftarrow t_0 + r$ leaks the union of the leakage of the first two assignments. More generally, the leakage of an arithmetic assignment is defined as the leakage of its two operands (with the convention that an input share $a[i]$ leaks $\{a[i]\}$).

The symbolic representation of leakage can be simplified by applying rules that preserve their semantics (defined formally in later sections). One commonly used rule is optimistic sampling, which replaces an expression of the form $e + r$, where $r$ only occurs once in the tuple, by $r$. We show the simplified leakage on the last column of Figure 1. Note that (simplified) leakage at each program point depends on at most one share of $a$ (either $a[0]$ or $a[1]$) and one share of $b$ (either $b[0]$ or $b[1]$). This suffices to conclude that the implementation is thus secure. We will make this claim precise in the next sections. For now, it suffices to get the following intuition: assume that $a$ and $b$ are the secrets, and $(a[0], a[1])$ is a secret sharing of $a$, i.e. $a[0]$ and $a[1]$ taken individually are uniformly distributed, and $a[0] + a[1] = a$. Then knowledge of $a[0]$ or $a[1]$ does not reveal any information about $a$. The situation is similar for $b$. Thus, knowledge of a single share of $a$ and a single share of $b$ does not reveal anything about them.

| Instruction | Symbolic value | Symbolic leakage | Simplified |
|---|---|---|---|
| $t_0 \leftarrow b[1] \times a[0]$ | $b[1] \times a[0]$ | $\{b[1], a[0]\}$ | $\{b[1], a[0]\}$ |
| $r \leftarrow_{\$} \{0,1\}$ | $r$ | $\{r\}$ | $\{r\}$ |
| $t_1 \leftarrow t_0 + r$ | $b[1] \times a[0] + r$ | $\{\mathbf{b[1]}, \mathbf{a[0]}, \mathbf{r}\}$ | $\{\mathbf{b[1]}, \mathbf{a[0]}, \mathbf{r}\}$ |
| $t_2 \leftarrow_{\mathtt{ff}} t_1$ | $b[1] \times a[0] + r$ | $\{b[1] \times a[0] + r\}$ | $\{r\}$ |
| $t_3 \leftarrow b[1] \times a[1]$ | $b[1] \times a[1]$ | $\{b[1], a[1]\}$ | $\{b[1], a[1]\}$ |
| $c[1] \leftarrow t_3 + t_2$ | $b[1] \times a + r$ | $\{\mathbf{b[1]}, \mathbf{a[1]}, \mathbf{b[1]} \times \mathbf{a[0]} + \mathbf{r}\}$ | $\{\mathbf{b[1]}, \mathbf{a[1]}, \mathbf{r}\}$ |
| $t_4 \leftarrow b[0] \times a[1]$ | $b[0] \times a[1]$ | $\{b[0], a[1]\}$ | $\{b[0], a[1]\}$ |
| $t_5 \leftarrow t_4 + r$ | $b[0] \times a[1] + r$ | $\{\mathbf{b[0]}, \mathbf{a[1]}, \mathbf{r}\}$ | $\{\mathbf{b[0]}, \mathbf{a[1]}, \mathbf{r}\}$ |
| $t_6 \leftarrow_{\mathtt{ff}} t_5$ | $b[0] \times a[1] + r$ | $\{b[0] \times a[1] + r\}$ | $\{r\}$ |
| $t_7 \leftarrow b[0] \times a[0]$ | $b[0] \times a[0]$ | $\{b[0], a[0]\}$ | $\{b[0], a[0]\}$ |
| $c[0] \leftarrow t_7 + t_6$ | $b[0] \times a + r$ | $\{\mathbf{b[0]}, \mathbf{a[0]}, \mathbf{b[0]} \times \mathbf{a[1]} + \mathbf{r}\}$ | $\{\mathbf{b[0]}, \mathbf{a[0]}, \mathbf{r}\}$ |

**Fig. 1.** Masked implementation of logical bit `and` against glitches. The second column contains the symbolic expression computed for each program point. The third and fourth columns are symbolic representations of leakage, before and after simplification. Maximal sets are written in bold. It is easy to check that $c[0] + c[1] = a \times b$.

Now consider the variant of the algorithm that omits the second flip-flop assignment in Figure 2. The leakage at the last program point is no longer independent of $a$, since both $a[0]$ and $a[1]$ appear in the tuple. Concretely, an attacker with access to the joint distribution $\{b[0], a[0], a[1], r\}$ can retrieve the second and third components and add them to obtain $a$.

We next describe how these examples are handled in our tool. The user provides a masked Verilog implementation of these algorithms and sets various parameters, including a security property (explained later). We first use an

| Instruction | Symbolic value | Symbolic leakage | Simplified |
|---|---|---|---|
| $t_0 \leftarrow b[1] \times a[0]$ | $b[1] \times a[0]$ | $\{b[1], a[0]\}$ | $\{b[1], a[0]\}$ |
| $r \leftarrow_{\$} \{0,1\}$ | $r$ | $\{r\}$ | $\{r\}$ |
| $t_1 \leftarrow t_0 + r$ | $b[1] \times a[0] + r$ | $\{\mathbf{b[1]}, \mathbf{a[0]}, \mathbf{r}\}$ | $\{\mathbf{b[1]}, \mathbf{a[0]}, \mathbf{r}\}$ |
| $t_2 \leftarrow_{\mathtt{ff}} t_1$ | $b[1] \times a[0] + r$ | $\{b[1] \times a[0] + r\}$ | $\{r\}$ |
| $t_3 \leftarrow b[1] \times a[1]$ | $b[1] \times a[1]$ | $\{b[1], a[1]\}$ | $\{b[1], a[1]\}$ |
| $c[1] \leftarrow t_3 + t_2$ | $b[1] \times a + r$ | $\{\mathbf{b[1]}, \mathbf{a[1]}, \mathbf{b[1]} \times \mathbf{a[0]} + \mathbf{r}\}$ | $\{\mathbf{b[1]}, \mathbf{a[1]}, \mathbf{r}\}$ |
| $t_4 \leftarrow b[0] \times a[1]$ | $b[0] \times a[1]$ | $\{b[0], a[1]\}$ | $\{b[0], a[1]\}$ |
| $t_5 \leftarrow t_4 + r$ | $b[0] \times a[1] + r$ | $\{\mathbf{b[0]}, \mathbf{a[1]}, \mathbf{r}\}$ | $\{\mathbf{b[0]}, \mathbf{a[1]}, \mathbf{r}\}$ |
| $t_6 \leftarrow b[0] \times a[0]$ | $b[0] \times a[0]$ | $\{b[0], a[0]\}$ | $\{b[0], a[0]\}$ |
| $c[0] \leftarrow t_5 + t_6$ | $b[0] \times a + r$ | $\{\mathbf{b[0]}, \mathbf{a[0]}, \mathbf{a[1]}, \mathbf{r}\}$ | $\{\mathbf{b[0]}, \mathbf{a[0]}, \mathbf{a[1]}, \mathbf{r}\}$ |

**Fig. 2.** Insecure masked implementation of logical bit `and` against glitches. The second column contains the symbolic expression computed for each program point. The third and fourth columns are symbolic representations of leakage, before and after simplification. Maximal sets are written in bold. It is easy to check that $c[0] + c[1] = a \times b$.

off-the-shelf tool (yosis) which generates an implementation in the `ilang` intermediate format (.ilang). The .ilang implementation is manually annotated to specify the `public` variables, the secret `input` variables, the secret `output` variables, and the `random` variables. Next, our tool generates from the annotated .ilang implementation an internal representation with a symbolic representation of leakage at each program point. At this point, verification starts. Our implementation exploits the fact that tuples of expressions are naturally ordered w.r.t the subset relation, e.g. the tuple $\{b[1], a[0], r\}$ leaks more than the singleton $\{b[1], a[0]\}$. Thus, it suffices to consider maximal leakage sets, which appear in bold in Figure 1. Whenever verification fails, i.e. a potentially flawed tuple is detected, our tool computes the joint distribution of this tuple, so as to verify exactly whether this tuple is an attack for the weakest security notion considered. This step is exact, therefore all false negatives are removed. Our tool successfully concludes for the secure examples, and outputs and checks the flawed tuple of intermediate computations for the insecure examples.

### 2.2 Transitions

For simplicity of exposition, we consider a model with transitions but no glitches (and thus do not use flip-flop gates). Figure 3 introduces another first-order masked implementation of logical `and`. The difference with the previous implementation is that variable $t_0$ is reused in the last but one instruction. As a consequence, observing the last but one instruction reveals both values of $t_0$, and depend on $b$. This is easily fixed by using a fresh variable $t_5$ in place of $t_0$. Interestingly, replacing $t_0$ with $t_5$ places us in a model in which every instruction leaks its symbolic expression, i.e. the abstract ISW model. In both cases, verification with our tool then proceeds as described in the previous subsection.

| Instruction | Symbolic value | Leakage |
|---|---|---|
| $t_0 \leftarrow b[1] \times a[0]$ | $b[1] \times a[0]$ | $\{b[1] \times a[0]\}$ |
| $r \leftarrow_\$ \{0,1\}$ | $r$ | $\{r\}$ |
| $t_1 \leftarrow t_0 + r$ | $b[1] \times a[0] + r$ | $\{b[1] \times a[0] + r\}$ |
| $t_2 \leftarrow b[1] \times a[1]$ | $b[1] \times a[1]$ | $\{b[1] \times a[1]\}$ |
| $c[1] \leftarrow t_1 + t_2$ | $b[1] \times a + r$ | $\{b[1] \times a + r\}$ |
| $t_3 \leftarrow b[0] \times a[1]$ | $b[0] \times a[1]$ | $\{b[0] \times a[1]\}$ |
| $t_4 \leftarrow t_3 + r$ | $b[0] \times a[1] + r$ | $\{b[0] \times a[1] + r\}$ |
| $t_0 \leftarrow b[0] \times a[0]$ | $b[0] \times a[0]$ | $\{b[1] \times a[0], b[0] \times a[0]\}$ |
| $c[0] \leftarrow t_4 + t_0$ | $b[0] \times a + r$ | $\{b[0] \times a + r\}$ |

**Fig. 3.** Masked implementation of logical bit `and` against transitions. The second column contains the symbolic expression computed for each program point. The third column contains leakage. It is easy to check that $c[0] + c[1] = a \times b$.

## 3 Programming model and security definitions

This section introduces an intermediate representation which captures different security models and notions, and presents algorithmic tools for checking that programs are secure. For the clarity of exposition, we focus on a simple setting without public variables. Adding public variables poses no technical difficulty, but clutters presentation.

### 3.1 Syntax and semantics of programs

Our intermediate representation abstracts away from the specifics of a particular security model, by requiring that all leakage is made explicit through program annotations. This eliminates the need to consider flip-flop assignments.

We assume throughout this paper that programs operate over Booleans. Figure 4 presents the syntax of programs as sequences of annotated instructions. An annotated instruction is an instruction annotated with a unique program point $p \in \mathcal{P}$, and a tuple $\ell$ of expressions which model its leakage. Instructions are probabilistic or deterministic assignments. We assume code to be written in 3-address form, i.e. the right-hand side of a deterministic assignment is of the form $v_1 + v_2$ or $v_1 \times v_2$, where $v_i$ is either a share $a[i]$, a deterministic variable $x$, or a probabilistic variable $r$. The left-hand side of an deterministic assignment is either a share $a[i]$ or a deterministic variable $x$. A probabilistic assignment is of the form $r \leftarrow_\$ \mathcal{K}$, where $r$ is drawn from a set $\mathcal{R}$ of probabilistic variables.

We now define the leakage. Let $\mathcal{L} = \bigcup_i \mathcal{K}^i$. For every discrete set $X$, $\mathsf{Distr}(X)$ denotes the set of distributions over $X$. A memory is a map that assigns to every share $a[i]$ a value in $\mathcal{K}$. We let $\mathcal{M}$ denote the set of memories. Now consider an observation set $O$, i.e. a subset of $\mathcal{P}$ such that $|O| \leq t$. We define the function:

$$[\![s]\!]_O : \mathcal{M} \to \mathsf{Distr}(O \to \mathcal{L})$$

which computes the joint leakage of $s$ on observation set $O$ on input memory $m \in \mathcal{M}$. The definition of $[\![s]\!]_O$ is obtained by pushing forward the instrumented

$$
\begin{array}{lll}
 & I ::= x \leftarrow v_1 \circ v_2 & \text{deterministic assignment} \\
v ::= r \mid x \mid a[i] & \mid\ a[i] \leftarrow v_1 \circ v_2 & \text{output assignment} \\
e ::= r \mid a[i] \mid e + e \mid e \times e & \mid\ r \leftarrow_{\$} \mathcal{K} & \text{probabilistic assignment} \\
\ell ::= \{e_1, \ldots, e_n\} & & \\
 & C ::= p : I \mid \ell & \text{instruction} \\
 & \mid\ C; C & \text{sequential composition}
\end{array}
$$

**Fig. 4.** Syntax of expressions, instructions and commands. $\circ$ ranges over $\{+, \times\}$. $x$ ranges over a set of deterministic variables $\mathcal{V}$, $r$ ranges over a set of probabilistic variables $\mathcal{R}$. $a[i]$ is called a share; $a$ is drawn from a set $\mathcal{A}$ and $i \in \{0, \ldots, t\}$ for some fixed value $t$, generally called order.

semantics $[\![s]\!] : \mathcal{M} \to \mathsf{Distr}(\mathcal{P} \to (\mathcal{K} \times \mathcal{L}))$ along the obvious projection function. The definition is standard, and omitted. The function $[\![s]\!]_O$ is naturally extended to distributions over memories; we abuse notation and still write $[\![s]\!]_O$.

### 3.2 Security notions

We recall three increasingly strong notions of security from the literature: probing security, threshold non-interference, and threshold strong non-interference. All notions capture some form of probabilistic non-interference.

*Probing security* is a notion of non-interference under uniform inputs. Formally, we define a set of universally uniform distributions and say that a command $s$ is $t$-probing secure iff for every observation set $O$ such that $|O| \leq t$ and universally uniform distributions $\mu$ and $\mu'$, we have $[\![s]\!]_O(\mu) = [\![s]\!]_O(\mu')$. Probing security considers a scenario where secret sharing is used to encode secret inputs, and the masked program is executed on encoded inputs. Since encodings are universally uniform, probing security entails that leakage does not depend on secrets.

For a concrete definition of universal uniformity, we consider the case of memories over inputs $a[0]$, $a[1]$, $b[0]$, and $b[1]$. In this setting, a distribution over memories is universally uniform iff it is the image of the function mapping pairs $(a, b)$ to the distribution

$$a_0 \leftarrow_{\$} \mathcal{K}; b_0 \leftarrow_{\$} \mathcal{K}; \mathsf{return}\ \langle a[0] \mapsto a_0, a[1] \mapsto a + a_0, b[0] \mapsto b_0, b_1 \mapsto b + b_0 \rangle$$

Probing security guarantees that leakage does not depend on secrets. Indeed, it is always possible to simulate leakage by generating an encoding of arbitrary values $a'$ and $b'$, and then executing the command on this encoding. This will result in an identical leakage.

*(Threshold) non-interference* can be understood as a notion of non-interference under cardinality constraints. A command $s$ is $t$-non-interfering ($t$-NI) if and only if any set of at most $t$ intermediate variables can be perfectly simulated from at most $t$ shares of each input. Concretely, a command $s$ is (threshold)

non-interfering at order $t$ iff for every observation set $O$ such that $|O| \leq t$, there exists an indexed family of sets $(I_a)_{a\in\mathcal{A}} \subseteq \{0,\ldots,t\}$ such that $|I_a| \leq t$ and for every initial memories $m$ and $m'$,

$$m \simeq_{(I_a)_{a\in\mathcal{A}}} m' \Longrightarrow [\![s]\!]_O(m) = [\![s]\!]_O(m')$$

where $m \simeq_{(I_a)_{a\in\mathcal{A}}} m'$ iff for every $a \in \mathcal{A}$ and $i \in I_a$, we have $m(a[i]) = m'(a[i])$. The intuition behind threshold non-interference is similar to the one behind probing security. In particular, threshold non-interference entails probing security.

For a realization of threshold non-interference, consider a masked implementation that takes as inputs $a[0]$, $a[1]$, $b[0]$, and $b[1]$. Threshold non-interference ensures that leakage only depends on one of the sets $\{a[i], b[j]\}$. Given that the secret $a$ is independent from $a[i]$ and similarly for $b$, it follows that leakage does not give any information about the secrets.

*(Threshold) strong non-interference [3]* is a very technical strengthening of (threshold) non-interference. It brings very strong composability guarantees that do not hold for (threshold) non-interference. Technically, strong non-interference imposes more stringent cardinality constraints. For every observation set $O$, we distinguish between internal observations $O_{in}$ (program points where the lhs of the assignment is a variable) and output observations $O_{out}$ (program points where the left-hand side of the assignment is a share $a[i]$). We say that a command $s$ is $t$-strong non-interfering ($t$-SNI) iff for every observation set $O$ such that $|O| \leq t$, there exists an indexed family of sets $(I_a)_{a\in\mathcal{A}} \subseteq \{0,\ldots,t\}$, such that $|I_a| \leq |O_{in}|$ and for every initial memories $m$ and $m'$,

$$m \simeq_{(I_a)_{a\in\mathcal{A}}} m' \Longrightarrow [\![s]\!]_O(m) = [\![s]\!]_O(m').$$

It is put forward in [19] that if a gadget is strongly non-interfering with glitches (which requires storing its outputs in flip flops so that they are stable and cannot propagate glitches), then the general composition rules introduced in [3] apply to hardware implementations with glitches. Being able to verify such stronger security notions is therefore helpful to analyze full ciphers and high number of shares, since it allows analyzing smaller (computationally tractable) parts of them independently, with global security guarantees thanks to composition.

## 4 Algorithmic verification

Checking probing or (S)NI security requires to verify a probabilistic non-interference property for all observation sets of size $t$. We define a generic verification parameterized by a test specific to each security property. The algorithm follows the same overall structure as `maskVerif` and relies on two functions. The first function Check is a *verification* algorithm for proving the probabilistic non-interference property of a fixed observation set. The function CheckAll is an

*exploration* algorithm which (adaptively) scans all the possible sets of observations. Verification succeeds if the algorithm proves absence of leakage for all observation sets.

To verify that an observation set $O$ (a tuple of expressions) is independent from some secret, the key idea is to apply successive transformation on $O$ into $O'$, preserving its distribution, until a termination condition Test is able to syntactically decide the independence. The Test function depends on the property:

- For probing security, we check if the tuple is independent from the initial mapping by checking syntactically if the secret inputs do not appear in $O'$.
- For non-interference, we check if for each input parameter $a$, at most $t$ shares $a[i]$ occur in the tuple $O'$.
- For strong non-interference, the condition is similar: for each parameter $a$, at most $|O_{in}|$ shares $a[i]$ should occur in $O'$.

The transformation of $O$ is based on optimistic sampling rule: if $r \notin e$ then $r$ and $e + r$ follow the same distribution, as well as $O$ and $O'$ where $r$ is replaced by $e+r$ ($O\{r \leftarrow e+r\}$). The condition $r \notin e$ (i.e $r$ is not a variable of $e$) ensures that the distributions of $r$ and $e$ are independent. The critical step is to select a substitution that will guarantee that the method terminates. Take for example $O = (r, x + r)$. If we replace $r$ by $x + r$, we obtain after simplification $(x + r, r)$ on which we could apply the same transformation again and again.

**Verification of single observation set.** The Check verification algorithm is summarized in Figure 5: it takes as input an observation set represented as a tuple $O$ of expressions. If Test is satisfied then Check succeeds. Otherwise, it uses the Select procedure to perform a transformation of $O$ into $O'$. To guarantee termination, the algorithm first attempts to check if $O$ can be rewritten (modulo associativity and commutativity of $+$) as $C[e + r]$ where $C[\cdot]$ is a context, and $r \notin e \cup C$, i.e. $r$ does not occur in $e$ and $C$). If it is the case, we apply the optimistic sampling rule and get $C[e + r]\{r \leftarrow e + r\} = C[e + (e + r)] = C[r]$. Notice that in that case the size of $C[r]$ is less than the size of $O$ (i.e the size of the resulting $O'$ decreases).

If the algorithm cannot exhibit such a context, it tries to apply the general optimistic sampling rule (removing the condition $r \notin C$). The resulting expression is the simplification of $O\{r \leftarrow e+r\}$. For the simplification, we basically use the ring laws but the distributivity makes harder the detection of new simplifications. Notice that this time the size of the resulting $O' = O\{r \leftarrow e+r\}$ does not necessarily decrease. To ensure termination, we add a set $R$ of random variables on which the general rule has already been used. The application of the rule is conditioned by the fact that $r \notin R$. The termination of the Check algorithm is ensured since either $R$ increases or the size of $O$ decreases (lexicographic order).

When more than one $r$ allow to apply the rules (i.e for the selection of the context), we define the multiplicative depth of a random variable and we rewrite in increasing order of multiplicative depth. For instance, in the expression $r + (r' + e) \times e'$ we assign multiplicative depth 0 to $r$ and 1 to $r'$.

Verification algorithm

```
proc Select(R, O) =
  if ∃r, e, C | O = C[e + r] ∧ r ∉ e ∪ C then
    return (R, (e, r), C[r]);
  if ∃r, e, C | O = C[e + r] ∧ r ∉ e ∪ R then
    O' = Simplify(O{r ← e + r});
    return (R ∪ {r}, (e, r), O');
  else fail ;
```

```
proc Check(R, B, O) =
  if Test(O) then return B;
  (R', b, O') = Select(R, O);
  Check(R', B :: b, O');
```

Exploration algorithm

```
proc Replay(B, O) =
  if B = [] then return Test(Simplify(O))
  if B = (e, r) :: B' then
    Replay(B', O{r ← e + r})
```

```
proc Extend(B, X) =
  {O ∈ X |
    Replay(B, O)}
```

```
proc OptSampling(X) =
  if ∃r, e, C_X | X = C_X[e + r] ∧ r ∉ e ∪ C_X then
    OptSampling(C_X[r]);
  else return X;
```

```
proc CheckAll(X) =
  if X = ∅ return true;
  X = OptSampling(X);
  O = Choose(X);
  B = Check(∅, [], O);
  X_0 = Extend(B, X);
  CheckAll(X \ X_0);
```

**Fig. 5.** Verification algorithm for probing security

We can prove that our new algorithm always terminates and is sound, i.e. it can detect all the attacks in our models. Note that considering only the first rule (first `if` statement of Select) makes our algorithm equivalent to the one of [2]. When we apply both rules (the two `if` statements of Select), our algorithm is equivalent to the one of [4], inspired from Gaussian elimination: contrary to this last one, we do not require the expressions to be linear. An additional advantage is the absence of false negatives when all the expressions are linear (completeness), it is no more the case if we remove the second `if` in Select.

Both algorithms return the list $B$ of optimistic sampling rules that have been applied: successive transformations in the exploration algorithm can be replayed.

**Exploration.** The exploration algorithm ensures that the verification algorithm analyzes all the possible sets of at most $t$ intermediate variables. However, rather than verifying each set separately, the exploration algorithm recursively checks larger sets, as in [2]. The idea behind the exploration algorithm is that if a tuple $O$ is probabilistic non-interferent then all sub-tuples of $O$ are. We present a very high level description of the algorithm to highlight the main differences with [2].

The algorithm CheckAll is presented in figure 5. Let $X$ be the set of all tuples that need to be checked. If $X$ is empty all tuples are trivially checked and the algorithm returns true. Else, it first tries to simplify as much as possible the set $X$ by applying the simple optimistic sampling rule, as in the first `if` of Select. This point is really crucial because it allows to share simplifications between all

tuples in $X$ and was not done in [2]. Then, the algorithm chooses an element $O$ in $X$ and tries to check it. If $O$ is successfully verified, the result $B$ is a list of optimistic sampling transformations that can be applied to prove independence of $O$. Next, the algorithm selects all the elements of $X$ that can be checked using the transformation $B$ using Extend[6]. At this point all elements in $X_0$ are known to satisfy the desired properties. Finally the algorithm needs to check the remaining tuples $X \setminus X_0$.

Initially, $X$ represents the set of all tuples of $t$ elements that can be generated within the set of $m$ possible observations. Its size is $\binom{m}{t}$. A naive implementation would thus be exponential in $t$ and it is crucial to have efficient data structures to represent $X$ and to implement the functions OptSampling, Extend, and $X \setminus X_0$. We use the data structures presented in [2] (worklist base space splitting).

Moreover, we use a representation of expressions as imperative graphs. This allows to detect easily if the simplest version of optimistic sampling rule can be applied (used in the first conditional of Select and OptSampling), and to compute efficiently the resulting expression.

## 5 Experiments

This section reports on experimental evaluation of our approach.

*Examples.* Our examples are mainly extracted from the available database provided by the authors of [9]. It gathers four different Verilog implementations of a masked multiplication. Three of them are implemented at the first masking order only, while DOM AND, designed in [22], is available up to order $t = 4$, i.e. when sensitive data is split into $t + 1 = 5$ shares. For the latter, we also consider modified versions that achieve non-interference and strong non-interference. Larger implementations are also provided, namely three S-boxes. AES S-box as designed in [22] and both versions of FIDES S-box as designed in [8] are implemented at the first order. We also consider a second-order and third-order AES S-box [21], and a Keccak S-box as designed in [23] and implemented from the first to the third order. In addition to this existing set of examples, Keccak S-box is analyzed at two extra orders, namely $t = 4$ and $t = 5$, and two versions of a different multiplication PARA AND [5] are verified from the first to the fourth order.

*Benchmarks.* First of all, we compare our tool with [2] which can only check probing security without glitches. While our tool is a variant, the resulting implementation is much more efficient. For example, checking probing security for the ISW multiplication at order 4 (resp. order 5) takes 1 (resp. 45) second using [2], while it takes only 0.1 (resp. 2.6) second with our tool.

Table 2 summarizes the verification outcome of the examples[7]. We use a 2.6 GHz Intel Core i7 with 16 GB of RAM running on macOS High Sierra, while Bloem et al. [9] use a Intel Xeon E5-2699v4 CPU with a clock frequency

[6] Missing tuples with Extend does not impact the correctness of the algorithm.

[7] Programs/logs are available at https://sites.google.com/view/submission-cav/home

of 3.6 GHz and 512 GB of RAM running in a 64-bit Linux OS environment. The table reports on verification for the three main security properties, namely SNI, NI, and probing security, and for two scenarios: a hardware scenario (HW) with glitches, and a software scenario (SW) without physical defaults. While the tool can also take into account transitions, we omit such examples as most of our implementations come from hardware where each wire is assigned only once (and so do not have transition).

The first column of the table (# obs) indicates the number of possible observations in the targeted implementation. In the software scenario, it corresponds to the number of intermediate variables. In the hardware scenario with glitches, it corresponds to the number of optimal observations. Note the latter is much lower than in the software scenario since non-maximal observation sets are ignored. Also note that while this first column displays the number of observations $n$ that will be further treated, verification at order $t$ requires the analysis of $\binom{n}{t}$ tuples. For instance, the verification of Keccak S-box in the software scenario at order 4 requires the analysis of $\binom{450}{4} \approx 2^{31}$ tuples. The second, third, and fourth columns report on the verification times in the 6 modes. We report $0.01s$ when the result is instantaneous and $\infty$ when the computations take more than 10 hours. When an implementation is insecure in a weaker model, then its verification time is equal for the stronger model. To report the outcome, a cross is displayed when a concrete attack is exhibited. Otherwise, the verification ends up successfully, indicating that the implementation is secure.

*Comparison with Bloem et al (EUROCRYPT 2018).* Bloem *et al.* [9] present a formal technique for proving security of implementations in the ISW model with glitches. Their method is based on Xiao-Massey lemma, which provides a necessary and sufficient condition for a boolean function to be statistically independent from a subset of its variables. Informally, the lemma states that a boolean function $f$ is statistically independent of a set of variables $X$ iff the so-called Fourier coefficients of every non-empty subset of $X$ is null. However, since the computation of Fourier coefficients is computationally expensive, they use instead an approximation method whose correctness is established in their paper. By encoding their approximation in logical form, they are able to instantiate their approach using SAT-based solvers. Their tool can verify implementations of AES, Keccak and FIDES S-Boxes, but the verification cost is significant.

The last column indicates the timings from [9] which are only available for probing security with and without glitches[8]. A dash is displayed when the example is not tested in [9]. The results show that our tool performs significantly better than the algorithm provided in [9]. For instance, the verification of the hardware first-order masked implementation of AES S-box is at the very least 7826 times much faster with our tool. In particular, note that some of the benchmarks provided for the tool of Bloem et al. only concern the verification of one secret (the ranking corresponds to the fastest and the lowest verification of the secrets). They are highlighted with a symbol $*$. As a consequence, without par-

[8] Note that the timings of [9] are obtained with a more powerful machine than ours.

allelization, these timings would probably be significantly higher. Our algorithm can also be parallelized (it is an option of our tool), but we only use this option for Keccak at order 5 since it makes the timing measurement less accurate.

## 6 Related work

This section reviews the state-of-the-art verification tools for software (without physical defaults but transitions) and hardware masked implementations.

**Software implementations.** Moss et al. [31] were the first to consider the use of automated methods to build or verify masked implementations. Specifically, they implement a type-based masking compiler that tracks which variables are masked by random values and iteratively modifies an unprotected program until all secrets are masked. This strategy ensures security against first-order DPA.

While type-based verification is generally efficient and scalable, it is also often overly conservative, i.e. it rejects secure programs. Logic-based verification often strikes interesting trade-offs between efficiency and expressiveness. This possibility was first explored in the context of masked implementations by Bayrak et al. [6]. Concretely, they propose a SMT-based method for analyzing the security of masked implementations against first-order DPA. Contrary to [31] which targets proving a stronger property of programs, their method directly targets proving statistical independence between secrets and leakage. While it is limited to first-order masking, it was extended to higher orders by Eldib, Wang and Schaumont [17]. Their approach is based on a logical characterization of security, akin to non-interference, and is based on model counting. While model counting incurs an exponential blow-up in the security order, and becomes infeasible even for relatively small orders, Eldib *et al.* circumvent the issue using incremental verification. Although such methods help, the scope of application of their methods remains limited. Recently, Zhang et al. [34] show how abstraction-refinement techniques provide significant improvement in terms of precision and scalability. Their tool, called SCInfer, alternates between fast and moderately precise approaches (partly inspired from [2] below) and computationally expensive but precise approaches.

Independently, Barthe et al. [2] propose a different approach for proving probing security. They establish and leverage a tight connection between the security of masked implementations and probabilistic non-interference, for which they propose efficient verification methods. Specifically, they show how a relational program logic previously used for mechanizing proofs of provable security can be specialized into an efficient procedure for proving probabilistic non-interference, and develop techniques that overcome the combinatorial explosion of observation sets for high orders. The concrete outcome of their work is the `maskVerif` framework, which achieves practicality at reasonably high orders, and prove security in all introduced non-interference security notions. A tweaked version additionally handles verification in presence of transitions, but hardware physical defaults

**Table 2.** Overview of verification of masked hardware circuits

| | # obs | | SNI | | NI | | probing | | probing [9] | |
|---|---|---|---|---|---|---|---|---|---|---|
| | HW | SW | HW | SW | HW | SW | HW | SW | HW | SW |
| first-order verification | | | | | | | | | | |
| Trichina AND [33] | 2 | 13 | 0.01s ✗ | 0.01s ✗ | 0.01s ✗ | 0.01s ✗ | 0.01s ✗ | 0.01s ✗ | $\leq$ 2s ✗ | $\leq$ 1s ✗ |
| ISW AND [25] | 1 | 13 | 0.01s ✗ | 0.01s | 0.01s ✗ | 0.01s | 0.01s ✗ | 0.01s | $\leq$2s ✗ | $\leq$1s |
| TI AND [32] | 3 | 21 | 0.01s ✗ | 0.01s ✗ | 0.01s | 0.01s | 0.01s | 0.01s | $\leq$3s | $\leq$1s |
| DOM AND [22] | 4 | 13 | 0.01s ✗ | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | $\leq$2s | $\leq$1s |
| DOM AND SNI | 6 | 13 | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | _ | _ |
| PARA AND [5] | 6 | 16 | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | _ | _ |
| DOM Keccak S-box [23] | 20 | 76 | 0.01s ✗ | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | $\leq$20s | $\leq$1s |
| DOM AES S-box [22] | 96 | 571 | 0.02s ✗ | 0.04s ✗ | 0.02s ✗ | 0.04s ✗ | 0.06s | 0.6s | $\leq$5-10h* | $\leq$30s* |
| TI Fides-160 S-box [8] | 192 | 6657 | 0.2s ✗ | 0.2s ✗ | 0.3s | 57s | 0.3s | 2.8s | $\leq$1-3s* | $\leq$1-2s* |
| TI Fides-192 APN [8] | 128 | 69281 | 2.3s ✗ | 2.46s ✗ | 2.25s | $\infty$ | 2.3s | 3m49s | $\leq$ 5s-2h | $\leq$2s-20m |
| second-order verification | | | | | | | | | | |
| DOM AND [22] | 12 | 30 | 0.01s ✗ | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | $\leq$1s | $\leq$1s |
| DOM AND SNI | 15 | 30 | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | _ | _ |
| PARA AND [5] | 15 | 30 | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | 0.01s | _ | _ |
| DOM Keccak S-box [23] | 60 | 165 | 0.01s ✗ | 0.2 | 0.07s | 0.14s | 0.03s | 0.03s | $\leq$40s* | $\leq$10s* |
| DOM AES S-box [21] | 168 | 1205 | 3s ✗ | 3m9s ✗ | 3s ✗ | 3m9s ✗ | 10.7s | 15m45s | _ | _ |
| third-order verification | | | | | | | | | | |
| DOM AND [22] | 20 | 54 | 0.01s ✗ | 0.04s | 0.02s | 0.05s | 0.02s | 0.03s | $\leq$20s | $\leq$4s |
| DOM AND SNI | 24 | 54 | 0.04s | 0.04s | 0.03s | 0.05s | 0.03s | 0.03s | _ | _ |
| PARA AND NI [5] | 20 | 48 | 0.01s ✗ | 0.01s ✗ | 0.02s | 0.03s | 0.02s | 0.02s | _ | _ |
| PARA AND SNI [5] | 28 | 53 | 0.04s | 0.05s | 0.02s | 0.04s | 0.02s | 0.02s | _ | _ |
| DOM Keccak S-box [23] | 100 | 290 | 0.01s ✗ | 41s | 3.6s | 11.6s | 0.49s | 0.68s | $\leq$25m* | $\leq$4m* |
| DOM AES S-box [21] | 296 | 2011 | 0.05s ✗ | 0.05s ✗ | 0.05s ✗ | 0.05s ✗ | 12m36s | $\infty$ | _ | _ |
| fourth-order verification | | | | | | | | | | |
| DOM AND [22] | 30 | 87 | 0.03s ✗ | 0.34s | 0.1s | 0.15s | 0.1s | 0.1s | $\leq$7m | $\leq$2m |
| PARA AND NI [5] | 35 | 75 | 0.01s ✗ | 0.01s ✗ | 0.15s | 0.42s | 0.18s | 0.15s | _ | _ |
| PARA AND SNI [5] | 40 | 85 | 0.34s | 0.81s | 0.17s | 0.47s | 0.16s | 0.16s | _ | _ |
| DOM Keccak S-box [23] | 150 | 450 | 0.02s ✗ | $\infty$ | 4m | 13m20 | 20s | 41s | _ | _ |
| fifth-order verification | | | | | | | | | | |
| DOM Keccak S-box [23] | 210 | 618 | 0.02s ✗ | $\infty$ | 1h6m | $\infty$ | 3m59s | 14m6s | _ | _ |

(e.g., glitches) are not supported. This work remains also permissive to false negatives. In the same line of work, Coron [13] presents an alternative tool, called `checkMasks`, which achieves similar functionalities as `maskVerif`, but exploits a more extensive set of transformations for operating on tuples of expressions. This is useful to achieve better verification times on selected examples.

A follow-up work by Barthe et al. [3] addresses the problem of compositional reasoning by introducing the notion of strong non-interference (SNI), and adapts `maskVerif` to check SNI. The adaptation achieves similar coverage as the original tool, i.e. it achieves practicality at reasonably high-orders. In addition, [3] proposes an information flow type system with cardinality constraints, which forms the basis of a compiler, called `maskComp`. This compiler transforms an unprotected implementation into an implementation that is protected at any desired order. Somewhat similar to the masking compiler of [31], `maskComp` uses typing information to control and to minimize the insertion of mask refreshing gadgets. In the same line of work, Belaïd, Goudarzi, and Rivain recently propose `tightPROVE` [7] which exactly and directly verifies the software probing security of a circuit based on standard gadgets at any order.

**Hardware implementations.** As recalled in the previous section, Bloem et al. [9] provide a tool for proving probing security of masked implementations in the ISW model with glitches. While this tool benefits from the new treatment of physical defaults, it faces efficiency issues and cannot handle classical higher-order examples. Recently Bloem, Iusupov, Krenn, and Mangard [10] provide some technical optimizations based on an earlier version of this submission (using our same tool), but that are still restricted to proofs on probing security. Namely, proven implementations thus cannot be safely composed to achieve larger secure ones. The work of Faust et al. follows the alternative approach of proving the strong non-intereference of some basic gadgets with glitches, which allows composing circuits at arbitrary orders (but less efficiently) [19].

## 7 Conclusions

We have developed and implemented an automated method for verifying masked implementations in presence of physical defaults. Our tool is based on novel and efficient algorithms for proving probabilistic non-interference for all admissible observation sets by an attacker. Our tool conveniently supports the three main notions of security (probing security, threshold non-interference and strong non-interference) and is able to verify efficiently implementations at high orders.

In the future, it would be interesting to extend our work beyond purely qualitative security definitions, and to consider quantitative definitions that upper bound how much leakage reveals about secrets — using total variation distance [18] or more recent metrics that directly or indirectly relate to noisy leakage security [15, 16]. More speculatively, it would also be interesting to extend our framework and verification methodologies to active adversaries, who

can tamper with computations [24]. A first step would be to extend the correspondence between information flow and simulation-based security to the case of active adversaries. An appealing possibility would be to exploit the well-known dual view of information flow security for confidentiality and integrity. It would also be interesting to build tools based on our algorithms to synthesize masked implementations.

## References


1. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 64–81, 2014.
2. G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, Apr. 2015.
3. G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini. Strong non-interference and type-directed higher-order masking. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 16*, pages 116–129. ACM Press, Oct. 2016.
4. G. Barthe, M. Daubignard, B. M. Kapron, Y. Lakhnech, and V. Laporte. On the equality of probabilistic terms. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 46–63. Springer, 2010.
5. G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 535–566. Springer, Heidelberg, Apr. / May 2017.
6. A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne. Sleuth: Automated verification of software power analysis countermeasures. In G. Bertoni and J.-S. Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 293–310. Springer, Heidelberg, Aug. 2013.
7. S. Belaïd, D. Goudarzi, and M. Rivain. Tight private circuits: Achieving probing security with the least refreshing. *IACR Cryptology ePrint Archive*, 2018:439, 2018.
8. B. Bilgin, A. Bogdanov, M. Knežević, F. Mendel, and Q. Wang. Fides: Lightweight authenticated cipher with side-channel resistance for constrained hardware. In G. Bertoni and J.-S. Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 142–158. Springer, Heidelberg, Aug. 2013.
9. R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter. Formal verification of masked hardware implementations in the presence of glitches. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 321–353. Springer, Heidelberg, Apr. / May 2018.
10. R. Bloem, R. Iusupov, M. Krenn, and S. Mangard. Sharing independence & relabeling: Efficient formal verification of higher-order masking. Cryptology ePrint Archive, Report 2018/1031, 2018. `https://eprint.iacr.org/2018/1031`.
11. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, Aug. 1999.
12. J. Coron, C. Giraud, E. Prouff, S. Renner, M. Rivain, and P. K. Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, pages 69–81, 2012.

13. J.-S. Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In B. Preneel and F. Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 65–82. Springer, Heidelberg, July 2018.
14. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 423–440. Springer, Heidelberg, May 2014.
15. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. *Journal of Cryptology*, 32(1):151–177, Jan. 2019.
16. A. Duc, S. Faust, and F.-X. Standaert. Making masking security proofs concrete - or how to evaluate the security of any leaking device. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 401–429. Springer, Heidelberg, Apr. 2015.
17. H. Eldib, C. Wang, and P. Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.
18. H. Eldib, C. Wang, M. M. I. Taha, and P. Schaumont. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(10):1558–1568, 2015.
19. S. Faust, V. Grosso, S. M. D. Pozo, C. Paglialonga, and F. Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.
20. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya. Koç, D. Naccache, and C. Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, pages 251–261. Springer, Heidelberg, May 2001.
21. H. Groß, M. Krenn, and S. Mangard. Second and third order verilog implementations of AES s-box, 2018.
22. H. Groß, S. Mangard, and T. Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In H. Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 95–112. Springer, Heidelberg, Feb. 2017.
23. H. Gross, D. Schaffenrath, and S. Mangard. Higher-order side-channel protected implementations of keccak. Cryptology ePrint Archive, Report 2017/395, 2017. `http://eprint.iacr.org/2017/395`.
24. Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner. Private circuits II: Keeping secrets in tamperable circuits. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 308–327. Springer, Heidelberg, May / June 2006.
25. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, Aug. 2003.
26. L. R. Knudsen and M. Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Springer, 2011.
27. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, Aug. 1996.
28. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, Aug. 1999.
29. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

30. S. Mangard, T. Popp, and B. M. Gammel. Side-channel leakage of masked CMOS gates. In A. Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, Heidelberg, Feb. 2005.
31. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 58–75. Springer, Heidelberg, Sept. 2012.
32. S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In P. Ning, S. Qing, and N. Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, Dec. 2006.
33. E. Trichina. Combinational logic design for AES subbyte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003. `http://eprint.iacr.org/2003/236`.
34. J. Zhang, P. Gao, F. Song, and C. Wang. Scinfer: Refinement-based verification of software countermeasures against side-channel attacks. In *Computer-Aided Verification*, 2018.